LiveCode Builder Language Reference

Introduction

LiveCode Builder is a variant of the current LiveCode scripting language (LiveCode Script) which has been designed for 'systems' building. It is statically compiled with optional static typing and direct foreign code interconnect (allowing easy access to APIs written in other languages).

Unlike most languages, LiveCode Builder has been designed around the idea of extensible syntax. Indeed, the core language is very small - comprising declarations and control structures - with the majority of the language syntax and functionality being defined in modules.

> **Note:** It is an eventual aim that control structures will also be extensible, however this is not the case in the current incarnation).

The syntax will be familiar to anyone familiar with LiveCode Script, however LiveCode Builder is a great deal more strict - the reason being it is intended that it will eventually be compilable to machine code with the performance and efficiency you'd expect from any 'traditional' programming language. Indeed, over time we hope to move the majority of implementation of the whole LiveCode system over to being written in LiveCode Builder.

> **Note:** One of the principal differences is that type conversion is strict - there is no automatic conversion between different types such as between number and string. Such conversion must be explicitly specified using syntax (currently this is using things like *... parsed as number* and *... formatted as string*.

Tokens

The structure of tokens is similar to LiveCode Script, but again a little stricter. The regular expressions describing the tokens are as follows:

- **Identifier**: [A-Za-z*][*A-Za-z0-9.]*
- **Integer**: [0-9]+
- **Binary Integer**: 0b[01]+
- **Hexadecimal Integer**: 0x[0-9a-fA-F]+
- **Real**: [0-9]+"."[0-9]+([eE][-+]?[0-9]+)?
- **String**: "[^\n\r"]*"
- **Separator**: Any whitespace containing at least one newline

Strings use backslash ('\') as an escape - the following are understood:

- **\n**: LF (ASCII 10)
- **\r**: CR (ASCII 13)
- **\t**: TAB (ASCII 9)
- **\q**: quote '"'
- **\u{X...X}: character with unicode codepoint U+X...X - any number of nibbles may be specified, but any values greater than 0x10FFFF will be replaced by U+FFFD.
- \: backslash '\'

> **Note:** The presence of '.' in identifiers are used as a namespace scope delimiter.
>
> **Note:** Source files are presumed to be in UTF-8 encoding.

Comments

LiveCode Builder supports single line comments, which begin with `//` or `--` and extend to the end of the line. There are also block comments, which begin with `/*` and end with `*/`, and can span multiple lines.

- **Single-line comment**: (--|//)[^\n\r]*
- **Block comment**: /*(^*?)\/

> **Note:** A block comment that spans multiple lines terminates the line of code that it begins on.

Line continuation

A LiveCode builder statement or declaration can be continued onto multiple lines of code by placing the line continuation character `\` at the end each line.

- **Line continuation**: \[\t]*(\n|\r\n|\r)

> **Note:** Tab and space characters are allowed after the `\` and before the newline, but no other characters.
>
> **Note:** A line continuation cannot occur within a comment.

Case-Sensitivity

At the moment, due to the nature of the parser being used, keywords are all case-sensitive and reserved. The result of this is that, using all lower-case identifiers for names of definitions should be avoided. However, identifiers *are* case-insensitive - so a variable with name pFoo can also be referenced as PFOO, PfOO, pfoO etc.

> **Aside:** The current parser and syntax rules for LiveCode Builder are constructed at build-time of the LiveCode Builder compiler and uses *bison* (a standard parser generator tool) to build the parser. Unfortunately, this means that any keywords have to be reserved as the parser cannot distinguish the use of an identifier in context (whether it is a keyword at a particular point, or a name of a definition).

It is highly recommended that the following naming conventions be used for identifiers:

- **tVar** - for local variables
- **pVar** - for in parameterns
- **rVar** - for out parameters
- **xVar** - for inout parameters
- **mVar** - for global variables in widgets
- **sVar** - for global variables in libraries
- **kConstant** - for constants
- Use identifiers starting with an uppercase letter for handler and type names.

By following this convention, there will not be any ambiguity between identifiers and keywords. (All keywords are all lower-case).

> **Note:** When we have a better parsing technology we will be evaluating whether to make keywords case-insensitive as well. At the very least, at that point, we expect to be able to make all keywords unreserved.

Typing

LiveCode Builder is a typed language, although typing is completely optional in most places (the only exception being in foreign handler declarations). If a type annotation is not specified it is simply taken to be the most general type *optional any* (meaning any value, including nothing).

The range of core types is relatively small, comprising the following:

- **nothing**: the single value *nothing*
- **Boolean**: one of *true* or *false*
- **Integer**: any integral numeric value (size limitations apply)
- **Real**: any numeric value (size and accuracy limitations apply)
- **Number**: any integer or real value
- **String**: a sequence of UTF-16 code units
- **Data**: a sequence of bytes
- **List**: a sequence of any values
- **Array**: a mapping from strings to values
- **any**: a value of any type

Additionally, all types can be annotated with **optional**. An optional annotation means the value may be the original type or nothing.

> **Note:** The current compiler does not do type-checking; all type-checking happens at runtime. However, this is being worked on so there will soon be a compiler which will give you type errors at compile-time.

Modules

```
Module
    : 'module' <Name: Identifier> SEPARATOR
        { ( Definition | Metadata | Import ) SEPARATOR }
      'end' 'module'
```

The smallest compilable unit of LiveCode Builder is the module. Each module is uniquely named using reverse DNS notation, and the names of modules are considered to live in a global namespace.

A module is a collection of public and private definitions, including constants, variables, types and handlers.

A module may depend on another module through import. An imported modules public definitions become accessible to the importing module.

> **Note:** For integration with the existing LiveCode system, there are two module variants which may be used. Widgets (use 'widget' instead of 'module') and Libraries (use 'library' instead of 'module'). A widget appears in LiveCode as a control, whilst a library adds all its public handlers to the bottom of the message path.

Metadata

```
Metadata
    : 'metadata' <Name: Identifier> 'is' <Value: String>
```

The metadata clauses allow a set of key-values to be encoded in the compiled module. These are not used in compilation or execution, but may be used by the system loading and using the module.

At the moment, the following keys are understood:

- title: a human-readable name for the module
- description: a simple description of the module's purpose
- version: a string in the form X.Y.Z (with X, Y and Z integers) describing the modules version
- author: the name of the author of the module
- os: the operating systems where the module can be used
- platforms: the platforms where the module can be used

> **Note:** The current metadata mechanism is unlikely to remain part of the language. It is intended that it will be replaced by a package description file, which will allow modules to be grouped together with other resources.

Imports

```
Import
    : 'use' <Name: Identifier>
```

The use clauses allow a module to refer to another module by importing all the target module's public definitions into its namespace.

The name of the module specified must be its full name, e.g. com.livecode.canvas.

A module may use any other module, as long as doing so does not cause a cycle in the dependency graph.

> **Note:** The current IDE and extension installation system does not yet implement arbitrary dependencies - the only dependencies it understands are those which are builtin to the system (e.g. com.livecode.canvas). However, you can still write and test out modules with dependencies locally - they just cannot be uploaded to the extensions portal.

Definitions

```
Definition
  : ( 'public' | 'private' ) ConstantDefinition
  | ( 'public' | 'private' ) TypeDefinition
  | ( 'public' | 'private' ) HandlerTypeDefinition
  | ( 'public' | 'private' ) VariableDefinition
  | ( 'public' | 'private' ) HandlerDefinition
  | ( 'public' | 'private' ) ForeignHandlerDefinition
  | PropertyDefinition
  | EventDefinition
```

Definitions are what are used to define usable entities in the language. All definitions are named using a unique identifier (so you cannot have two definitions with the same name).

Definitions can be either *public* or *private* (the default is private

- so there is no need to explicitly specify that). Public definitions are available when the module is used by another module whereas private definitions can only be used within the module.

> **Note**: Properties and events are, by their nature, always public as they define things which only make sense to access from outside.
>
> **Note**: When writing a library module, all public handlers are added to bottom of the message path in LiveCode Script.

Constants

```
ConstantDefinition
  : 'constant' <Name: Identifier> is <Value: Expression>
```

A constant definition defines a named constant. The value can be any expression which depends on only on constant values to evaluate.

Types

```
TypeDefinition
  : 'type' <Name: Identifier> 'is' <TypeOf: Type>
```

A type definition defines an alias, it names the given type with the given Name, allowing the name to be used instead of the type.

```
Type
  : <Name: Identifier>
  | 'optional' <Target: Type>
  | 'any'
  | 'nothing'
  | 'Boolean'
  | 'Integer'
  | 'Real'
  | 'Number'
  | 'String'
  | 'Data'
  | 'Array'
  | 'List'
  | 'Pointer'
```

A type clause describes the kind of value which can be used in a variable or parameter.

If a type is an identifier, then this is taken to be a named type defined in a type definition clause.

An optional type means the value can be either the specified type or nothing. Variables which are of optional type are automatically initial zed to nothing.

The remaining types are as follows:

- **any**: any value
- **Boolean**: a boolean value, either the value *true* or *false*.
- **Integer**: any integer number value
- **Real**: any real number value
- **Number**: any number value
- **String**: a sequence of UTF-16 code units
- **Data**: a sequence of bytes
- **Array**: a map from string to any value (i.e. an associative array, just like in LiveCode Script)
- **List**: a sequence of any value
- **nothing**: a single value *nothing* (this is used to describe handlers with no return value - i.e. void)
- **Pointer**: a low-level pointer (this is used with foreign code interconnect and shouldn't be generally used).

> **Note:** *Integer* and *Real* are currently the same as *Number*.
>
> **Note:** In a subsequent update you will be able to specify lists and arrays of fixed types. For example, *List of String*.
>
> **Note:** In a subsequent update you will be able to define record types (named collections of values - like structs in C) and handler types (allowing dynamic handler calls through a variable - like function pointers in C).

Handler Types

```
HandlerTypeDefinition
  : [ 'foreign' ] 'handler' 'type' <Name: Identifier> '(' [ ParameterList ] ')' [ 'retur
ns' <ReturnType: Type> ]
```

A handler type definition defines a type which can hold a handler. Variables of such types can hold handlers (just like function pointers in C) which allows them to be called dynamically.

If the handler type is defined as foreign then automatic bridging to a C function pointer will occur when the type appears as the type of a parameter in a foreign handler definition.

> **Note:** Passing an LCB handler to a foreign function requires creation of a function pointer. The lifetime of the function pointer is the same as the widget or module which created it.

Record Types

```
RecordTypeDefinition
  : 'record' 'type' <Name: Identifier> SEPARATOR
    { RecordTypeFieldDefinition }
    'end' 'record'

RecordTypeFieldDefinition
  : <Name: Identifier> [ 'as' <TypeOf: Type> ]
```

A record type definition defines a type that consists of 0 or more named fields, each with its own optional type.

Variables

```
VariableDefinition
  : 'variable' <Name: Identifier> [ 'as' <TypeOf: Type> ]
```

A variable definition defines a module-scope variable. In a widget module, such variables are per-widget (i.e. instance variables). In a library module, there is only a single instance (i.e. a private global variable).

The type specification for the variable is optional, if it is not specified the type of the variable is *optional any*meaning that it can hold any value, including being nothing.

Variables whose type has a default value are initialized to that value at the point of definition. The default values for the standard types are:

- **optional**: nothing
- **Boolean**: false
- **Integer**: 0
- **Real**: 0.0
- **Number**: 0
- **String**: the empty string
- **Data**: the empty data
- **Array**: the empty array
- **List**: the empty list
- **nothing**: nothing

Variables whose type do not have a default value will remain unassigned and it is a checked runtime error to fetch from such variables until they are assigned a value.

Handlers

```
HandlerDefinition
  : [ 'unsafe' ] 'handler' <Name: Identifier> '(' [ ParameterList ] ')' [ 'returns' <Ret
urnType: Type> ] SEPARATOR
      { Statement }
    'end' 'handler'
```

Handler definitions are used to define functions which can be called from LiveCode Builder code, invoked as a result of events triggering in a widget module, or called from LiveCode Script if public and inside a library module.

There is no distinction between handlers which return a value and ones which do not, apart from the return type. Handlers can be called either in expression context, or in statement context. If a handler which returns no value (it is specified as *returns nothing*) is called in expression context then its value is *nothing*.

```
ParameterList
  : { Parameter , ',' }

Parameter
  : ( 'in' | 'out' | 'inout' ) <Name: Identifier> [ 'as' <ParamType: Type>
```

The parameter list describes the parameters which can be passed to the handler. Handlers must be called with the correct number of parameters, using expressions which are appropriate to the mode.

An in parameter means that the value from the caller is copied to the parameter variable in the callee handler.

An out parameter means that no value is copied from the caller, and the value on exit of the callee handler is copied back to the caller on return.

> **Note:** It is a checked runtime error to return from a handler without ensuring all non-optional 'out' parameters have been assigned a value. However, this will only occur for typed variables whose type does not have a default value as those which do will be default initialized at the start of the handler.

An inout parameter means that the value from the caller is copied to the parameter variable in the callee handler on entry, and copied back out again on exit.

The type of parameter is optional, if no type is specified it is taken to be *optional any* meaning it can be of any type.

> **Note:** Only assignable expressions can be passed as arguments to inout or out parameters. It is a checked compile-time error to pass a non-assignable expression to such a parameter.

If 'unsafe' is specified for the handler, then the handler itself is considered to be unsafe, and may only be called from other unsafe handlers or unsafe statement blocks.

Foreign Handlers

```
ForeignHandlerDefinition
  : 'foreign' 'handler' <Name: Identifier> '(' [ ParameterList ] ')' [ 'returns' <Return
Type: Type> ) ] 'binds' 'to' <Binding: String>
```

A foreign handler definition binds an identifier to a handler defined in foreign code.

The last parameter in a foreign handler declaration may be '...' to indicate that the handler is variadic. This allows binding to C functions such as sprintf.

Note: No bridging of types will occur when passing a parameter in the non-fixed section of a variadic argument list. You must ensure the arguments you pass there are of the appropriate foreign type (e.g. CInt, CDouble).

There are a number of types defined in the foreign, java and objc modules which map to the appropriate foreign type when used in foreign handler signatures.

There are the standard machine types (defined in the foreign module):

- Bool maps to an 8-bit boolean
- Int8/SInt8 and UInt8 map to 8-bit integers

- Int16/SInt16 and UInt16 map to 16-bit integers
- Int32/SInt32 and UInt32 map to 32-bit integers
- Int64/SInt64 and UInt64 map to 64-bit integers
- IntSize/SIntSize and UIntSize map to the integer size needed to hold a memory size
- IntPtr/SIntPtr and UIntPtr map to the integer size needed to hold a pointer
- NaturalSInt and NaturalUInt map to 32-bit integers on 32-bit processors and 64-bit integers on 64-bit processors
- NaturalFloat maps to the 32-bit float type on 32-bit processors and the 64-bit float (double) type on 64-bit processors

There are the standard C primitive types (defined in the foreign module)

- CBool maps to 'bool'
- CChar, CSChar and CUChar map to 'char', 'signed char' and 'unsigned char'
- CShort/CSShort and CUShort map to 'signed short' and 'unsigned short'
- CInt/CSInt and CUInt map to 'signed int' and 'unsigned int'
- CLong/CSLong and CULong map to 'signed long' and 'unsigned long'
- CLongLong/CSLongLong and CULongLong map to 'signed long long' and 'unsigned long long'
- CFloat maps to 'float'
- CDouble maps to 'double'

There are types specific to Obj-C types (defined in the objc module):

- ObjcObject wraps an obj-c 'id', i.e. a pointer to an objective-c object
- ObjcId maps to 'id'
- ObjcRetainedId maps to 'id', and should be used where a foreign handler argument expects a +1 reference count, or where a foreign handler returns an id with a +1 reference count.

Note: When an ObjcId is converted to ObjcObject, the id is retained; when an ObjcObject converted to an ObjcId, the id is not retained. Conversely, when an ObjcRetainedId is converted to an ObjcObject, the object takes the +1 reference count (so does not retain); when an ObjcObject is put into an ObjcRetainedId, a +1 reference count is taken (so does retain).

There are aliases for the Java primitive types (defined in the java module)

- JBoolean maps to Bool
- JByte maps to Int8
- JShort maps to Int16
- JInt maps to Int32
- JLong maps to Int64
- JFloat maps to Float32
- JDouble maps to Float64

All the primitive types above will implicitly bridge between corresponding high level types:

- CBool and Bool bridge to and from Boolean
- All integer and real types bridge to and from Number

Other LCB types pass as follows into foreign handlers:

- any type passes an MCValueRef
- nothing type passes as the null pointer
- Boolean type passes an MCBooleanRef
- Integer type passes an MCNumberRef

- Real type passes an MCNumberRef
- Number type passes an MCNumberRef
- String type passes an MCStringRef
- Data type passes an MCDataRef
- Array type passes an MCArrayRef
- List type passes an MCProperListRef

Finally, the Pointer type passes as void * to foreign handlers. If you want a pointer which can be null, then use optional Pointer - LCB will throw an error if there is an attempt to map from the null pointer value to a slot with a non-optional Pointer type.

Modes map as follows:

- in mode is just pass by value
- out mode passes a pointer to a variable of one of the above types, the variable is uninitialized on entry
- inout mode passes a pointer to a variable of one of the above types, the variable is initialized to a value on entry

If the return type is of a Ref type, then it must be a copy.

If an out parameter is of a Ref type, then it must be a copy (on exit)

If an inout parameter is of a Ref type, then its existing value must be released, and replaced by a copy (on exit).

The binding string for foreign handlers is language-specific and currently supported forms are explained in the following sections.

Foreign handlers' bound symbols are resolved on first use and an error is thrown if the symbol cannot be found.

Foreign handlers are always considered unsafe, and thus may only be called from unsafe context - i.e. from within an unsafe handler, or unsafe statement block.

The C binding string

The C binding string has the following form:

```
"c:[library>][class.]function[!calling][?thread]"
```

Here *library* specifies the name of the library to bind to (if no library is specified a symbol from the engine executable is assumed).

Here *class* is currently unused.

Here *function* specifies the name of the function symbol to bind to (on Windows, the symbol must be unadorned, and so exported from the library by listing it in a DEF module).

Here *calling* specifies the calling convention which can be one of:

- `default`
- `stdcall`
- `thiscall`
- `fastcall`
- `cdecl`

- `pascal`
- `register`

All but `default` are Win32-only, and on Win32 `default` maps to `cdecl`. If a Win32-only calling convention is specified on a non-Windows platform then it is taken to be `default`.

Here *thread* is either empty or `ui`. The `ui` form is used to determine whether the method should be run on the UI thread (currently only applicable on Android and iOS).

The Obj-C binding string

The Obj-C binding string has the following form:

```
"objc:[library>][class.](+|-)method[?thread]"
```

Here *library* specifies the name of the library or framework to bind to (if no library is specified a symbol from the engine executable or a library it is linked to is assumed).

Here *class* specifies the name of the class containing the method to bind to. If the method is an instance method, the class can be omitted, creating a 'dynamic binding', i.e. just resolving the selector.

Here *method* specifies the method name to bind to in standard Obj-C selector form, e.g. addTarget:action:forControlEvents:. If the method is a class method then prefix it with '+', if it is an instance method then prefix it with '-'.

Here *thread* is either empty or `ui`. The `ui` form is used to determine whether the method should be run on the UI thread (currently only applicable on Android and iOS).

The Java binding string

The Java binding string has the following form:

```
"java:[className>][functionType.]function[!calling][?thread]"
```

Here *className* is the qualified name of the Java class to bind to.

Here *functionType* is either empty, or `get` or `set`, which are currently used for getting and setting member fields of a Java class.

For example

```
"java:java.util.Calendar>set.time(J)"
"java:java.util.Calendar>get.time()J"
```

are binding strings for setting and getting the `time` field of a Calendar object.

Here *function* specifies the name of the method or field to bind to. The function `new` may be used to call a class constructor. *function* also includes the specification of function signature, according to the standard rules for forming these when calling the JNI.

The function `interface` may be used on Android to create an interface proxy - that is an instance of a generic Proxy class for a given interface. This effectively allows LCB handlers to be registered as the targets for java interface callbacks, such as event listeners.

The foreign handler binding to such a function takes a value that should either be a `Handler` or an `Array` - if it is a `Handler`, the specified listener should only have one available callback. If the listener has multiple callbacks, an array can be used to assign handlers to each. Each key in the array must match the name of a callback in the listener. The specified handlers must match the callback's parameters and return type, using JObject where primitive type parameters are used.

Overloaded methods in the interface are not currently supported.

For example:

```
handler type ClickCallback(in pView as JObject) returns nothing

foreign handler _JNI_OnClickListener(in pHandler as ClickCallback) returns JObject binds
to "java:android.view.View$OnClickListener>interface()"

foreign handler _JNI_SetOnClickListener(in pButton as JObject, in pListener as JObject)
 returns nothing binds to "java:android.view.View>setOnClickListener(Landroid/view/View
$OnClickListener;)V"

public handler ButtonClicked(in pView as JObject) returns nothing
    post "buttonClicked"
    MCEngineRunloopBreakWait()
end handler

public handler SetOnClickListenerCallback(in pButton as JObject)
    unsafe
        variable tListener as JObject
        put _JNI_OnClickListener(ButtonClicked) into tListener
        _JNI_SetOnClickListener(pButton, tListener)
    end unsafe
end handler
```

or

```
handler type MouseEventCallback(in pMouseEvent as JObject) returns nothing

foreign handler _JNI_MouseListener(in pCallbacks as Array) returns JObject binds to "jav
a:java.awt.event.MouseListener>interface()"

foreign handler _JNI_SetMouseListener(in pJButton as JObject, in pListener as JObject) r
eturns nothing binds to "java:java.awt.Component>addMouseListener(Ljava/awt/event/MouseL
istener;)V"

public handler MouseEntered(in pEvent as JObject) returns nothing
    post "mouseEnter"
    MCEngineRunloopBreakWait()
end handler

public handler MouseExited(in pEvent as JObject)
    -- do something on mouse entter
end handler

public handler SetMouseListenerCallbacks(in pJButton as JObject)
    variable tArray as Array
    put MouseEntered into tArray["mouseEntered"]
    put MouseExited into tArray["mouseExited"]
    unsafe
        variable tListener as JObject
        put _JNI_MouseListener(tArray) into tListener
        _JNI_SetMouseListener(pJButton, tListener)
    end unsafe
end handler
```

> *Important:* On Android, interface callbacks are *always* run on the engine thread. This means JNI local references from other threads (in particular the UI thread) are unavailable. Therefore it is not advised to do anything using the JNI in interface callbacks.

Here *calling* specifies the calling convention which can be one of:

- `instance`
- `static`
- `nonvirtual`

Instance and nonvirtual calling conventions require instances of the given Java class, so the foreign handler declaration will always require a Java object parameter.

Here, *thread* is either empty or `ui`. The `ui` form is used to determine whether the method should be run on the UI thread (currently only applicable on Android and iOS).

> **Warning:** At the moment it is not advised to use callbacks that may be executed on arbitrary threads, as this is likely to cause your application to crash.

Properties

```
PropertyDefinition
  : 'property' <Name: Identifier> 'get' <Getter: Identifier> [ 'set' <Setter: Identifier
> ]
```

Property definitions can only appear in widget modules. They define a property which can be accessed from LiveCode Script in the usual way (e.g. *the myProperty of widget 1*).

Both getter and setter clauses can use either a variable or handler identifier. If a variable identifier is used, then the property value is fetched (and stored) from that variable. If a handler identifier is used then a handler is called instead.

A getter handler must take no arguments and return a value. A setter handler must take a single argument and return no value.

The set clause is optional. If it is not present then the property is read-only.

Events

```
EventDefinition
  : 'event' <Name: Identifier> '(' [ ParameterList ] ')' [ 'as' <ReturnType: Type> ]
```

Event definitions define a callable handler which calls back to the environment.

Note: Whilst events can be defined they currently cannot be used. To send a message to the LiveCode Script environment use the *dispatch* command which allows sending messages to arbitrary LiveCode Script objects.

Statements

```
Statement
  : VariableStatement
  | IfStatement
  | RepeatStatement
  | ThrowStatement
  | ReturnStatement
  | PutStatement
  | SetStatement
  | GetStatement
  | CallStatement
  | BytecodeStatement
  | UnsafeStatement
```

There are a number of built-in statements which define control flow, variables, and basic variable transfer. The remaining syntax for statement is defined in auxiliary modules.

Variable Statements

```
VariableStatement
  : 'variable' <Name: Identifier> [ 'as' <TypeOf: Type> ]
```

A variable statement defines a handler-scope variable. Such variables can be used after the variable statement and up to the end of the current statement block, but not before.

Variables whose type have a default value are initialized with that value at the point of definition in the handler. See the main Variables section for the defaults of the standard types.

> **Note:** It is a checked runtime error to attempt to use a variable whose type has no default before it is assigned a value.

The type specification for the variable is optional, if it is not specified the type of the variable is *optional any*meaning that it can hold any value, including being nothing.

If Statements

```
IfStatement
  : 'if' <Condition: Expression> 'then' SEPARATOR
      { Statement }
    [ { 'else' 'if' <Condition: Expression> 'then' SEPARATOR
        { Statement } ]
    [ 'else' SEPARATOR
        { Statement } ]
    'end' 'if'
```

The if statement enables conditional execution based on the result of an expression which evaluates to a boolean.

Each block of code in an if/else if/else statement defines a unique scope for handler-local variable definitions.

> **Note:** It is a checked runtime error to use an expression which does not evaluate to a boolean in any condition expression.

Repeat Statements

```
RepeatStatement
  : RepeatHeader SEPARATOR
      { Statement }
    'end' 'repeat'
  | 'next' 'repeat'
  | 'exit' 'repeat'

RepeatHeader
  : 'repeat' 'forever'
  | 'repeat' <Count: Expression> 'times'
  | 'repeat' 'while' <Condition: Expression>
  | 'repeat' 'until' <Condition: Expression>
  | 'repeat' 'with' <Counter: Identifier> 'from' <Start: Expression> ( 'up' | 'down' )
'to' <Finish: Expression> [ 'by' <Step: Expression> ]
  | 'repeat' 'for' 'each' <Iterator> 'in' <Container: Expression>
```

The repeat statements allow iterative execute of a sequence of statements.

The block of code present in a repeat statement defines a unique scope for handler-local variable definitions.

The **repeat forever** form repeats the body continually. To exit the loop, either an error must be thrown, or **exit repeat** must be executed.

The **repeat times** form repeats the body Count number times. If Count evaluates to a negative integer, it is taken to be zero.

> **Note:** It is a checked runtime error to use an expression not evaluating to a number as the Count.

The **repeat while** form repeats the body until the Condition expression evaluates to false.

> **Note:** It is a checked runtime error to use an expression not evaluating to a boolean as the Condition.

The **repeat until** form repeats the body until the Condition expression evaluates to true.

> **Note:** It is a checked runtime error to use an expression not evaluating to a boolean as the Condition.

The **repeat with** form repeats the body until the Counter variable reaches or crosses (depending on iteration direction) the value of the Finish expression. The counter variable is adjusted by the value of the Step expression on each iteration. The start, finish and step expressions are evaluated before the loop starts and are not re-evaluated. The Counter variable must be declared before the repeat statement.

> **Note:** It is a checked runtime error to use expressions not evaluating to a number as Start, Finish or Step.

The **repeat for each** form evaluates the Container expression, and then iterates through it in a custom manner depending on the Iterator syntax. For example:

```
repeat for each char tChar in "abcdef"
  -- do something with tChar
end repeat
```

The **next repeat** statement terminates the current iteration of the enclosing loop and starts the next iteration of the loop, or exits if currently on the last iteration.

The **exit repeat** statement terminates the current iteration of the enclosing loop, continuing execution at the statement after the enclosing loop's **end repeat**.

Throw Statements

```
ThrowStatement
  : 'throw' <Error: Expression>
```

The **throw** statement causes an error to be raised. This causes execution to terminate, and the error is passed back to environment.

The Error expression must be an expression that evaluates to a string.

> **Note:** There is currently no try / catch mechanism in LiveCode Builder, so throwing an error will cause the error to be raised in LiveCode Script in the appropriate context.

Return Statements

```
ReturnStatement
  : 'return' [ <Value: Expression> ]
```

The **return** statement causes execution of the current handler to end, and control return to the caller.

If a Value expression is specified, it is evaluated and returned as the result of the handler call.

> **Note:** It is a checked runtime error for a value returned from a handler to not match the return type of the handler it is in.

Transfer Statements

```
PutStatement
  : 'put' <Value: Expression> into <Target: Expression>

SetStatement
  : 'set' <Target: Expression> 'to' <Value: Expression>
```

The **put** and **set** statements evaluate the Value expression and assign the resulting value to the Target expression.

The target expression must be assignable.

> **Note:** It is a checked runtime error for the source value's type to not be compatible with the target expression's type.

```
GetStatement
  : 'get' <Value: Expression>
```

The **get** statement evaluates the Value expression and returns it as the result of the statement. The value is subsequently available by using **the result** expression.

Call Statements

```
CallStatement
  : <Handler: Identifier> '(' [ <Arguments: ExpressionList> ] ')'
```

The call statement executes a handler with the given arguments.

The Handler identifier must be bound to either a handler or foreign handler definition.

The Arguments are evaluated from left to right and passed as parameters to the variable.

> **Note:** It is a checked runtime error for the types of 'in' and 'inout' arguments to not match the declared types of the handler's parameters.

Any parameters of 'out' type are not evaluated on entry, but assigned to on exit.

Any parameters of 'inout' type are evaluated on entry, and assigned on exit.

> **Note:** It is a checked compile-time error to pass non-assignable expressions to parameters which are of either 'out' or 'inout' type.

The return value of a handler is subsequently available by using **the result** expression.

> **Note:** All handlers return a value, even if it is nothing. This means that calling a handler will always change **the result**.

Bytecode Statements

```
BytecodeStatement
  : 'bytecode' SEPARATOR
        { BytecodeOperation }
    'end'

BytecodeOperation
  : <Label: Identifier> ':'
  | 'register' <Name: Identifier> [ 'as' <Type: Type> ]
  | <Opcode: Identifier> { BytecodeArgument , ',' }

BytecodeArgument
  : ConstantValueExpression
  | Identifier
```

The bytecode statement allows bytecode to be written directly for the LiveCode Builder Virtual Machine.

Bytecode operation arguments can either be a constant expression, the name of a definition in current scope, the name of a register, or the name of a label in the current bytecode block. The exact opcodes and allowed arguments are defined in the LiveCode Builder Bytecode Reference.

Labels are local to the current bytecode block, and can be used as the target of one of the jump instructions.

Register definitions define a named register which is local to the current bytecode block. Registers are the same as handler-local variables except that they do not undergo default initialization.

Bytecode statements are considered to be unsafe and can only appear inside unsafe handlers or unsafe statement blocks.

> **Note:** Bytecode blocks are not intended for general use and the actual available operations are subject to change.

Unsafe Statements

```
UnsafeStatement
  : 'unsafe' SEPARATOR
        { Statement }
    'end' 'unsafe'
```

The unsafe statement allows a block of unsafe code to be written in a safe context.

In particular, calls to unsafe handlers (including all foreign handlers) and bytecode blocks are allowed in unsafe statement blocks but nowhere else.

Expressions

```
Expression
  : ConstantValueExpression
  | VariableExpression
  | ResultExpression
  | ListExpression
  | ArrayExpression
  | CallExpression
```

There are a number of expressions which are built-in and allow constant values, access to call results, list construction and calls. The remaining syntax for expressions is defined in auxiliary modules.

Constant Value Expressions

```
  ConstantValueExpression
    : 'nothing'
    | 'true'
    | 'false'
    | INTEGER
    | REAL
    | STRING
```

Constant value expressions evaluate to the specified constant value.

The **nothing** expression evaluates to the nothing value and can be assigned to any optional typed variable.

The **true** and **false** expressions evaluate to boolean values.

The INTEGER and REAL expressions evaluate to numeric values.

The STRING expression evaluates to a string value.

Constant value expressions are not assignable.

Variable Expressions

```
VariableExpression
  : <Name: Identifier>
```

Variable expressions evaluate to the value of the specified variable.

Variable expressions are assignable.

Result Expressions

```
ResultExpression
  : 'the' 'result'
```

The result expression evaluates to the return value of the previous (executed) non-control structure statement.

Result expressions are not assignable.

List Expressions

```
ListExpression
  : '[' [ <Elements: ExpressionList> ] ']'
```

A list expression evaluates all the elements in the expression list from left to right and constructs a list value with them as elements. Each expression is converted to `optional any` when constructing the list.

The elements list is optional, so the empty list can be specified as *[]*.

List expressions are not assignable.

Array Expressions

```
ArrayExpression
  : '{' [ <Contents: ArrayDatumList> ] '}'
ArrayDatumList
  : <Head: ArrayDatum> [ ',' <Tail: ArrayDatumList> ]
ArrayDatum
  : <Key: Expression> ':' <Value: Expression>
```

An array expression evaluates all of the key and value expressions from left to right, and constructs an **Array**value as appropriate. Each key expression must evaluate to a **String**. Each value expression is converted to `optional any` when constructing the array.

The contents are optional, so the empty array can be written as `{}`.

Array expressions are not assignable.

Call Expressions

```
CallExpression
  : <Handler: Identifier> '(' [ <Arguments: ExpressionList> ] ')'
```

A call expression executes a handler.

Its use is identical to a call statement, except that the return value of the handler is the value of the expression, rather than being available as **the result**.

> **Note:** Handlers which return no value (i.e. have nothing as their result type) can still be used in call expressions. In this case the value of the call is **nothing**.

Namespaces

Identifiers declared in a module are placed in a scope named using the module name. This allows disambiguation between an identifier declared in a module and an identical one declared in any of its imports, by using a fully qualified name.

For example:

```
module com.livecode.module.importee

public constant MyName is "Importee"
public handler GetMyName() returns String
    return MyName
end handler

public type MyType is Number

end module

module com.livecode.module.usesimport

use com.livecode.module.importee

public constant MyName is "Uses Import"
public handler GetMyName() returns String
    return MyName
end handler
public type MyType is String

handler TestImports()
    variable tVar as String
    put MyName into tVar
    -- tVar contains "Uses Import"

    put com.livecode.module.importee.MyName into tVar
    -- tVar contains "Importee"

    put com.livecode.module.usesimport.MyName into tVar
    -- tVar contains "Uses Import"

    put com.livecode.module.importee.GetMyName() into tVar
    -- tVar contains "Importee"

    variable tVarMyType as MyType
    put tVar into tVar1 -- valid

    variable tVarImportedType as com.livecode.module.importee.MyType
    put tVar into tVarImportedType -- compile error
end handler

end module
```

Experimental Features

Warning: This section describes current language features and syntax that are considered experimental and unstable. They are likely to change or go away without warning.

Safe Foreign Handlers

```
SafeForeignHandler
  : '__safe' 'foreign' 'handler' <Name: Identifier> '(' [ ParameterList ] ')' [ 'return
s' <ReturnType: Type> ) ] 'binds' 'to' <Binding: String>
```

By default foreign handlers are considered unsafe and thus can only be used in unsafe blocks, or unsafe handlers. However, at the moment it is possible for a foreign handler to actually be safe if it has been explicitly written to wrap a foreign function so it can be easily used from LCB.

Specifically, it is reasonable to consider a foreign handler safe if it conforms to the following rules:

- Parameter types and return type are either ValueRefs, or bridgeable types
- Return values of ValueRef type are retained
- If the function fails then MCErrorThrow has been used
- 'out' mode parameters are only changed if the function succeeds
- A return value is only provided if the function succeeds

Examples of foreign handlers which can be considered safe are all the foreign handlers which bind to syntax in the LCB standard library.

Foreign Aggregate Types

C-style aggregates (e.g. structs) can now be accessed from LCB via the new aggregate parameterized type. This allows calling foreign functions which has arguments taking aggregates by value, or has an aggregate return value.

Aggregate types are foreign types and can be used in C and Obj-C foreign handler definitions. They bridge to and from the List type, allowing an aggregate's contents to be viewed as a sequence of discrete values.

Aggregate types are defined using a `foreign type` clause and binding string. e.g.

```
public foreign type NSRect binds to "MCAggregateTypeInfo:qqqq"
```

The structure of the aggregate is defined by using a sequence of type codes after the ':', each type code represents a specific foreign (C) type:

Char	Type
a	CBool
b	CChar
c	CUChar
C	CSChar

Char	Type
d	CUShort
D	CSShort
e	CUInt
E	CSInt
f	CULong
F	CSLong
g	CULongLong
G	CSLongLong
h	UInt8
H	SInt8
i	UInt16
I	SInt16
j	UInt32
J	SInt32
k	UInt64
K	SInt64
l	UIntPtr
L	SIntPtr
m	UIntSize
M	SIntSize
n	Float
N	Double
o	LCUInt
O	LCSInt
p	NaturalUInt

Char	Type
P	NaturalSInt
q	NaturalFloat
r	Pointer

When importing an aggregate to a List, each field in the aggregate is also bridged, except for Pointer types which are left as Pointer. When exporting an aggregate from a List, each element is bridged to the target field type.

Note: Any foreign type binding to an aggregate must be public otherwise the type will not work correctly.